Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1837 – 1846

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml☆

M. Danelutto[a], R. Di Cosmo[b]

[a]*Dept. Computer Science, Univ. of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy*
[b]*Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, INRIA Paris-Rocquencourt, F-75205 Paris, France*

Abstract

We discuss the implementation of a minimalist parallel library in OCaml. The library provides parallel map and fold (reduce) higher order functions and targets standard cache coherent shared memory multi-cores. Our `Parmap.parmap` and `Parmap.parfold` functions may be used to seamlessly replace OCaml `List map` and `fold` standard functions preserving their full functional semantics while achieving nearly optimal speedup on standard multi-core architectures. We discuss the design of the `Parmap` module, the main implementation features and we present some experimental results assessing the efficiency of the `Parmap` parallel functions. Overall, `Parmap` represents a perfect incarnation of the "propagate the concept with minimal disruption" principle introduced in Cole's algorithmic skeleton *manifesto*.

Keywords: structured parallel programming, algorithmic skeletons, map, reduce

1. Introduction

Multi-cores with some kind of shared memory subsystem constitute the *de facto* standard computing elements in a wide range of systems, from mobile phones and tablets to workstations and servers, up to high end parallel computer systems. However, the development of efficient applications for these inherently parallel systems still requires a consistent design and programming effort. Classical programming techniques exploiting threads and the associated cooperation and synchronization machinery represent a kind of "assembly level parallel language" for multi-cores, and inherit all the problems of low level languages: writing efficient multithreaded code is difficult and error prone, debugging and maintenance are even more problematic. The adoption of slightly higher level programming paradigms such as OpenMP improves programmer productivity with respect to plain Pthreads, but still requires a consistent expertise to tune all the available parameters in such a way that good efficiency is achieved.

Several authors suggested to fill the evident gap between programmability and available parallelism in multi-cores with different kinds of structured programming models: parallel design patterns [1] as in [2, 3] or algorithmic skeletons [4] as in [5, 6, 7, 8, 9], just to name a few examples. Algorithmic skeletons, in particular, promise to raise the level of abstraction provided to parallel application programmers by making available mechanisms—language constructs,

☆This work has been partially supported by EU FP7 STREP Project ParaPhrase, and partially performed at IRILL, `http://www.irill.org`.
Email addresses: `marcod@di.unipi.it` (M. Danelutto), `roberto@dicosmo.org` (R. Di Cosmo)

doi:10.1016/j.procs.2012.04.202

objects, components or plain library calls—that completely encapsulate and manage a parallelism exploitation pattern. The application programmer writing a parallel application is therefore only left with the duty of picking up and instantiating the proper (composition of) skeletons modelling the parallelism at hand from a library of predefined, optimized and, possibly, portable algorithmic skeletons.

The price to pay with algorithmic skeletons is the necessity for the programmers to "learn" new concepts—the algorithmic skeleton (parallel) semantics—in order to be able to write parallel applications.

Actually, the "*four principles which we believe should guide the future design and the development of skeletal programming systems*" stated in Cole's "skeleton manifesto" [4] included the principle stated as "*propagate the concept with minimal conceptual disruption*". In other words, the principle states algorithmic skeletons should be introduced in such a way that programmers are not required to do things far from their normal programming activity or requiring brand new knowledge and competences.

In this work, we present an experiment aiming at providing parallel algorithmic skeletons to OCaml programmers fully respecting Cole's minimal disruption principle: we provide a lightweight OCaml library that presents to the OCaml programmer a simple replacement of the basic operations on lists provided by the standard library, namely `List.map` and `List.fold_right`/`List.fold_left` that implement the now well known map and reduce paradigm[1]. Our replacement functions implement the map and fold operations in parallel, using the existing cores of the processor at hand, in a very efficient and clean way.

The advantage is twofold: on the one hand, OCaml programs with heavyweight map, reduce, and map–reduce computations may be seamlessly improved by just changing the name of the library calls used, e.g. invoking our `Parmap.parmap` instead of the classic `List.map` one; on the other hand, a simple speedup model is exposed to the OCaml programmers–use `Parmap` calls any time you want to improve independent data parallel or reduce style computation execution times–that may be taken into account when designing new complex and time consuming applications with many items to be processed independently.

The rest of this paper outlines the `Parmap` design issues and choices (Sec. 2), provides a detailed description of its current implementation (Sec. 3), reports several experimental results assessing its feasibility and efficiency (Sec. 4) and finally discusses the relevant related work (Sec. 5).

2. Parmap design

Suitable strategies for the parallel computation of map and reduce are well known. The computation of a map requires applying the same function to all the elements in a list:

$$map\ f\ [x_1; \ldots; x_k] = [f\ x_1; \ldots; f\ x_k]$$

In a pure map, all the $(f\ x_i)$ computations are independent and therefore any partition of the result list may be computed in parallel. The result of a *reduce* $\oplus$ applied to a list is

$$reduce\ \oplus\ [x_1; \ldots; x_k] = x_1 \oplus \ldots \oplus\ x_k$$

The $\oplus$ function is required to be associative and commutative. If we imagine the reduce computation as a tree of $\oplus$ applications, any sub tree rooted at a node at level j may be computed in parallel and the partial results of the computations of the sub trees may be then "summed up" with a sequential application of $\oplus$ to compute the final reduce result.

The idea of `Parmap` is to exploit the cores available on modern architectures to compute in parallel maps and reduces (folds) using the following steps:

Step 1 The input list is split into a number of partitions which may be specified by the programmer or default to the number of available cores.

Step 2 Each partition is independently computed. In case of a map, each element x_i is transformed using the "worker" function f into $(f\ x_i)$. In case of a reduce, the partition elements $x_j, \ldots, x_{j+k}$ are summed up using the worker function $\oplus$ into $x_j \oplus \ldots \oplus x_{j+k}$.

[1] This is the reason why we'll use the terms *fold* and *reduce* basically as synonyms from now on

STEP 3 The partial results are combined into the final result. This happens sequentially. In case of a map, the sub lists are appended; in case of a reduce, the partial results are summed up using again the $\oplus$ function.

As far as reduce is concerned, this computing schema corresponds to first computing in parallel all the y_j defined as

$$y_1 = x_1 \oplus \ldots \oplus x_{n_1} \qquad y_2 = x_{n_1+1} \oplus \ldots \oplus x_{n_2} \qquad \ldots \qquad y_{k-1} = x_{n_k+1} \oplus \ldots \oplus x_n$$

that is the reduces computed on the k different partitions, and then computing sequentially the final result

$$y_1 \oplus y_2 \oplus \ldots \oplus y_{k-1}$$

Therefore steps 1 and 3 should be performed as fast as possible as they represent the "serial fraction" of the algorithm and it is their duration that will eventually limit the final speedup according to the well known Amdahl law. Sec. 3 will detail the steps performed in these two phases to minimize the overheads.

As far as the user interface of `Parmap` is concerned, we chose to implement the "minimal disruption" principle by providing a `Parmap` through a standard OCaml module with an interface as close as possible to that of the `List` module. The `List` module provides the `map` and `fold_right` functions with the following signatures:

```
List.map : ('a -> 'b) -> 'a list -> 'b list
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

The signature of the corresponding `Parmap` functions is instead:

```
type 'a sequence = L of 'a list | A of 'a array

val parmap : ?ncores:int -> ?chunksize:int ->
             ('a -> 'b) -> 'a sequence -> 'b list
val parfold: ?ncores:int -> ?chunksize:int ->
             ('a -> 'b -> 'b) -> 'a sequence -> 'b -> ('b->'b->'b) -> 'b
```

If we disregard the optional parameters, the signature of the `List.map` is the same of the signature of `Parmap.parmap`, but for the fact the "collection" processed is not an `'a list` but rather it is an `'a sequence`. We used the `'a sequence` to uniformly support the possibility to process both lists and arrays.

The optional parameter `ncores` has been included to control the number of cores to be used, while the `chunksize` parameter has been included to control the parallelism grain. In order to get a parallel version on n cores of a program with a computationally expensive

```
let res = List.map f l
```

it is therefore sufficient to change the code into

```
let res = Parmap.parmap ~ncores:n f (Parmap.L l)
```

This will run achieving a speedup very close to the number of cores on the target architecture[2] (see Sec. 4).

The `Parmap.parfold` has a signature which is the same one of the `List.fold_right` but for: i) the optional parameters (that can be obviously omitted), ii) the `'a sequence` used instead of the `'a list`, for the very same reasons discussed for the `map`, and iii) the additional "combine" operator with type `'b -> 'b -> 'b`. This last parameter is the only one real, significant difference with the `List.fold_right`. It has been introduced to be able to specify two different $\oplus$ operators: one for the reduce computed in parallel on the different partitions (this is the first `'a -> 'b -> 'b` parameter) and one for the reduce operator needed to accumulate sequentially the final `parfold` result out of the partial reduce results computed on the different partitions (the last `'b -> 'b -> 'b`) parameter). This may be useful in some particular cases. However, if the user wants to parallelize an expression such as

```
(List.fold_right oplus list oplus_zero)
```

[2] Of course, this is true if the *whole* computation of the program is represented by the `res map`. If more computations are performed, these will be not affected by the `Parmap` and therefore the overall speedup will be smaller according to the Amdhal law.

it is sufficient to substitute it by the expression

```
(Parmap.parfold oplus (Parmap.L list) oplus_zero oplus)
```

It's worth mentioning that when it is used *without* the load balancing optimization described in section 3.1, `Parmap.parfold` actually does not require that the $\oplus$ operator(s) be commutative: in that case, the fold computed in parallel on the different partitions are internally sequential, and composed in the same order, so commutativity is not used (lines 25-29 of Figure 1).

When the load balancing optimization is used, however, commutativity *is* required, as usual in the implementations of fold operations that seek to achive performance using asynchronous mechanisms that force out-of-order evaluation on the elements of the input data.

3. Parmap implementation

The first choice we had to make when implementing `Parmap` was between the two main mechanisms available to run in parallel the workers of the second step: processes or threads. Although threads are normally the standard choice on multi-core architectures, we were forced to use processes because in the current implementation of OCaml, all threads are run concurrently on the same core, so no speedup can be achieved using threads.

Our process-based implementation requires some additional effort for step 1 and 3, but also provides an interesting encapsulation of side effects: they are not propagated to the global state when the results are built after computing a map, thus enforcing the embarrassingly parallel pattern modelled by this kind of map.

Having chosen processes as the way of implementing step 2, we moved on to implement step 1 and 3, trying to minimize their overhead to maximize speedup. Figure 1 shows the essential parts of the OCaml code used for the simplest form of implementation of our library.

Step 1. In step 1, (lines 3 to 9 in the code) a number of processes is forked according to a user provided parameter[3].

Step 2. After the fork, each child process (lines 11 to 16 in the code) inherits the state of the parent automatically, so it can determine the interval `lo...hi` of the data it must handle, by using the local value of the `i` index (lines 11 to 13). The computation is then performed in line 14 on the specified segments using the function `compute` that is built out of the user function according to the operation to perform (map, reduce or map/reduce). Finally, in line 15, the result is marshalled into a shared memory area associated to a file descriptor indexed by `i`.

Step 3. After all the children have finished (line 21), the parent collects the list of the results from the children by unmarshaling the data contained in the shared memory areas associated to each child (lines 25-17), and combines them in line 29 using a function `combine` that is also built out of the user function according to the operation to perform (map, reduce or map/reduce).

This schema allows to provide `Parmap` functions processing either lists or arrays, and this is the reason why in Sec. 2 the `Parmap` signature exports the `'a sequence` type definition and all generic `Parmap` entries accept `'a sequences`.

As already stated, step 1 and step 3 represent the serial parts of our implementation, and they should be carefully optimized in order to be able to achieve suitable speedups. Step 1 mainly relies on the fact that the `fork` is very efficient and in modern operating system manages to copy *on demand* the pages that are written to by the child process. This means that in our case–child process read their own data partition and actually only write the results of the computation on the input partition–the overhead paid is fairly negligible. We made some experiments aimed at measuring the overhead incurred in the `fork` phase of the child processes. The time in between the start of the `Parmam.parmap` call and the start of the activities of the forked child processes is of the order of milliseconds (5-25 msecs). It's worth to point out that this time includes all the setup activities needed to prepare the communication channels in between the forking process and the forked child processes, which includes quite a number of system calls. We also measured the time between the end of the parallel partition processing and the end of the `Parmap.parmap`

[3] In the real implementation, this defaults to the number of cores in the target architecture (derived through proper library calls).

```
1  let simplemapper ncores compute opid al combine =
2    (* init task parameters *)
3    let ln = Array.length al in
4    let chunksize = ln/ncores in
5    (* create descriptors to mmap *)
6    let fdarr=Array.init ncores (fun _ -> tempfd()) in
7    (* spawn children *)
8    for i = 0 to ncores-1 do
9      match Unix.fork() with
10       0 -> (* children code: compute on the chunk *)
11            (let lo=i*chunksize in
12             let hi=if i=ncores-1 then ln-1
13                    else (i+1)*chunksize-1 in
14             let v = compute al lo hi opid in
15             marshal fdarr.(i) v;
16             exit 0)
17     | -1 -> failwith "Fork error"
18     | pid -> ()
19   done;
20   (* wait for all children *)
21   for i = 0 to ncores-1 do ignore(Unix.wait()) done;
22   (* read in all data *)
23   let res = ref [] in
24   (* accumulate the results in the right order *)
25   for i = 0 to ncores-1 do
26       res:= ((unmarshal fdarr.((ncores-1)-i)):'d)::!res;
27   done;
28   (* combine all results *)
29   combine !res;;
```

Figure 1: Simple implementation of the distribution, fork, and recollection phases in `Parmap`

call, that is the time needed to implement Step 3 in the schema detailed above. In our experiments, using the machines detailed in Sec. 4, we verified that Step 3 takes milliseconds to complete too (2-3 msecs). At the end of Sec. 4 we will comment more on the impact of such "serial fraction" times.

3.1. Load balancing and other optimizations

The simple implementation described above has one major drawback: it does not allow the grain of the parallel computation to be specified. Indeed, we can only use it with a number of chunks equal to the number of cores (the chunksize is computed in line 4 from the length of the sequence). If the chunks are big and their computation cost is non homogeneous, this can severely hinder performance.

This is why the library also contains a more sophisticated implementation that refines `simplemapper` using a bidirectional channel between the main process and the children. Each child requests (the index of) a chunk, performs the computation and accumulates the result locally, and repeats this cycle until the parent signals that the computation is complete; at that point, it returns the result in the shared memory area, exactly as in the case of the `simplemapper` function above. This algorithm corresponds to the well known on demand schema for process farms, that achieves automatic load balancing [5]. It is used whenever the chunksize specified by the user forces more than one chunk on a core.

The implementation of the `Parmap` exported functions can then be written as follows, where `mapper` calls either the `simplemapper` shown above, or the on-demand implementation we just described, according to the value of `chunksize`

```
let parmap ?ncores ?chunksize (f:'a -> 'b) (s:'a sequence) : 'b list=
  let al = match s with A al -> al | L l  -> Array.of_list l in
  let compute al lo hi previous exc_handler =
    let f' j = f (Array.unsafe_get al (lo+j)) in
    let rec aux acc =
        function 0 ->  (f' 0)::acc
               | n ->  aux ((f' n)::acc) (n-1)
    in aux previous (hi-lo)
  in
  mapper ncores ~chunksize compute [] al  (fun r -> List.concat r)
```

We also considered the issue of the kernel scheduler moving some of the processes in charge of computing in parallel `Parmap` chunks from one core to another. As different core groups usually share cache levels, if the process is moved to a different group of cores cache usage turns out to be inefficient. `Parmap` therefore implements process *pinning*. The processes created to process `Parmap` partitions are forced to be executed on exactly one core among those available. This mechanisms lowers the number of cache faults, and therefore improves the general efficiency of `Parmap` (see Sec. 4).

3.2. Special optimizations for manipulating arrays of floats

Another performance bottleneck comes from the need to perform marshaling and unmarshaling of the results between the parent and the children processes: when using a strongly and statically typed programming language, like OCaml, this operation is unavoidable unless one knows the precise type of the return data, *and* its exact size.

But there is one important situation in which we actually know the size and the type of the result: the mapping of a floating point operation on large arrays of floats. This particular configuration is common enough in numerical code to deserve a special treatment, and our library provides a highly optimised function array_float_parmap that avoids the marshalling overhead completely. We highlight here these optimisations.

To understand the efficiency issues involved when parallelizing a map computation on a large array of floats, it is useful to examine the different steps that an implementation of a parallel map function may need to perform.

1. create an array to hold the result of the computation. This operation involves an initialisation phase and can be expensive: on an Intel i7, creating an array of 10 million floats takes 50 milliseconds;
2. create a shared memory area for returning the result;
3. copy the result array to the shared memory area;
4. perform the computation in the children writing the result in the shared memory area;
5. copy the result back to the OCaml array;

Steps 1, 2 and 4 are unavoidable: the return array and the a shared memory area for returning the result must be created (one might merge these two operations into one, but without a significant cost saving), and of course one needs to perform the required computation. On the other side, steps 3 and/or 5 may be omitted depending on what the user wants to do with the result:

- using precise knowledge on the layout of arrays of floats in OCaml, one can access the shared memory area as if it was an array of floats, without need to copy the result array into this area (this is what our code does);
- it may be safe to leave the result in the shared memory area, to be reused by later calls, but for this one needs to give up the benefit of automatic garbage collection on the result array, as the content of the shared memory area is not garbage collected (we decided agains this solution in our code).

The array_float_parmap function in our library performs only the steps 1, 2, 4 and 5. These optimizations allow to reduce the overhead of the parallel map operation quite significantly with respect to the stock implementation of parmap on arrays.

It is also possible for the user to share the steps 1, 2 among subsequent calls to array_float_parmap by preallocating the result array and the shared memory buffer, and passing them as optional parameters; this may save a significant amount of time if the array is very large and the operation is repeated frequently.

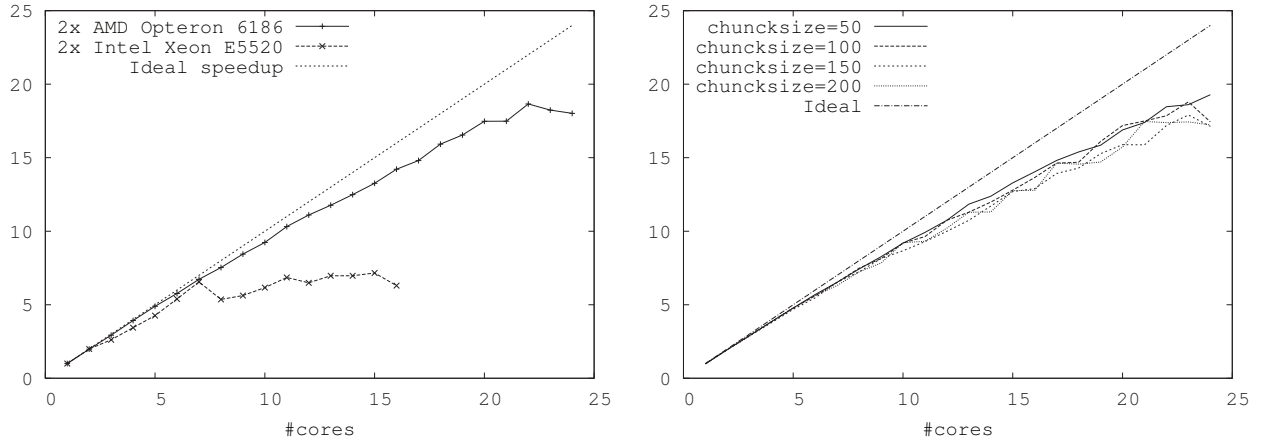

Figure 2: Sample `Parmap` speedup (left: `Parmap` with no optional parameters, right: with different chunksize values)

4. Experiments

We run a number of experiments aimed at validating the `Parmap` design. Actually, these experiments contributed to understanding the reason of different overheads in the first `Parmap` versions and to understanding which were the most effective ways to get rid of them. It's worth pointing out that our `Parmap` library has been published quite early as open source: as a consequence, we received comments and suggestions from different OCaml parallel programmers that helped improve the efficiency of `Parmap`.

The experiments have been run on several different architectures, including Intel i3 and i7 CPUs and more aggressive dual Xeon E5520 (8 cores in total) and dual AMD Opteron 6176 (24 cores in total) CPUs. All the results reported in this Section have been achieved using synthetic applications.

Fig. 2 shows speedups achieved when computing a map with a simple synthetic worker function on integers in an application made of a single map call. The plot on the left shows speedup obtained just by changing a `map f l` with a `Parmap.map f (Parmap.L l)`. The plot on the right shows slightly better results in the execution of the same application on the AMD Opteron. It is related to the very same application where we manually specified a chunksize for the `Parmap`. As evidenced by the different curves, however, finding the correct chunksize value is not in general an easy task.

Typical speedups do not change significantly in case computations performed floating point operations are considered, although in this case hyper threading on Xeon is less effective, due to the scarce duplication of floating point resources. Fig. 3 left shows the speedup of an application with the same structure of the one used in Fig. 2 but using a floating point number crunching worker function.

The amount of computation performed on the single item is anyway significant, as shown in Fig. 3 right: the higher the amount of time spent in the computation of the single element, the better the speedup, as expected. In this case the processing of the whole list (20K elements) requires a sequential time as shown in the legend. Therefore the time spent in computing the single element is in the three cases 27.5μsecs, 270.5μsecs and 2.75msecs, respectively. This latter value leads to almost linear speedup.

These performance figures may be better appreciated if we take into account the overheads measured for the `Parmap.parmap` serial fraction discussed in Sec. 3. With times spent in the serial fraction of the algorithm in the range of milliseconds, it is clear that we can only obtain good speedups when the overall amount of time spent in the sequential computation makes these milliseconds negligible. Amdahl law states the maximum speedup we can achieve is $\frac{1}{f}$, with f representing the serial fraction of the algorithm. Therefore, to achieve a speedup of n–the parallelism degree of our architecture–we should have a serial fraction smaller than $\frac{1}{n}$. In our case, this means that when our sequential time is in the range of half a second ($Tseq = 0.55secs$ in the right plot of Fig. 3) and considered the serial

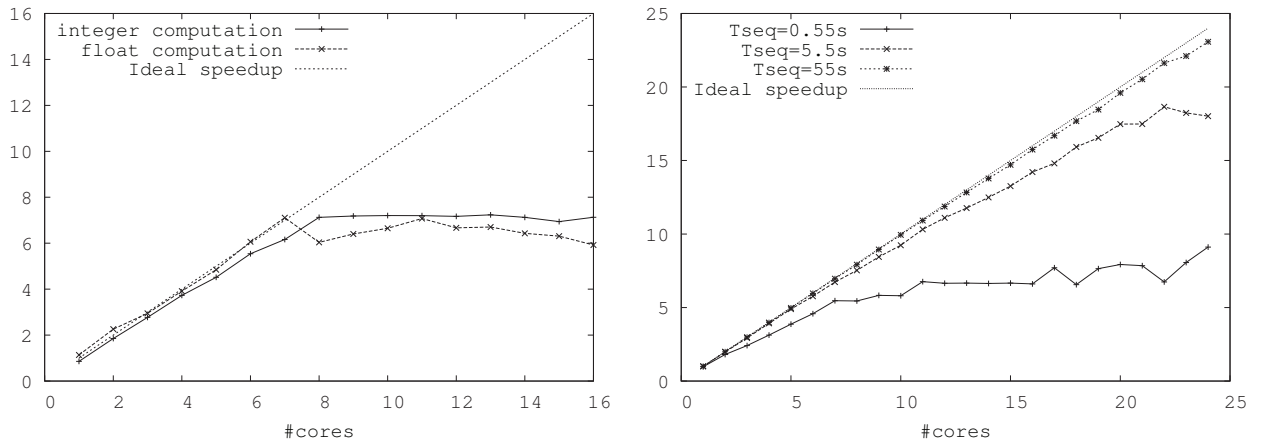

Figure 3: `Parmap` speedup (int vs floating point computations on Intel Xeon E5520) (left) and effect of grain (on AMD Opteron) (right)

overhead (Step 1 plus Step 3) exceeds the tens of milliseconds, we will definitely not be able to achieve the expected speedup values on the 24 core architecture.

`Parmap.parfold` behaviour is not different from the one demonstrated by the `Parmap`. Fig. 4 (left) shows the speedups achieved on the AMD 6176 Opteron architecture when parallelizing a program made of a single fold call by substituting the `List.fold_right` with a `Parmap.parfold` without any optional parameters. Speedups do not change when using `Parmap` functions operating on arrays, instead of lists (see Fig. 4 right), as expected.

Fig. 5 shows the effects of thread pinning. The left plot is relative to the application running sequentially in 55 secs. These application scales pretty decently on the 24 core architecture. In this case the effect of pinning is evident. The right plot is instead relative to the synthetic application running sequentially in about 5.5 seconds. In that case, as evidenced also in Fig. 3 the application efficiency is much lower. In these cases pinning actually do not add any kind of benefit, although it does not even introduce any sensible additional overhead.

Since we are proposing a "minimalistic" parallel skeleton library for Ocaml, we were more interested in demonstrating the different features of the library rather than showing speedups achieved in real applications, that almost never turn out to be just a single call to a map or to a reduce function. This is why all the experiments whose results have been discussed in this Section are relative to execution of *synthetic* benchmarks, and not real applications.

The actual speedup measured running real applications with calls to `Parmap` functions will be greatly affected by the particular mix of `Parmap` and non `Parmap` code and could be hardly used to evaluate the efficiency of the `Parmap` implementation. Nevertheless, based on the feedback we received from actual users of the library, we can say that real applications actually benefit from the usage of `Parmap` proportionally to the amount of time spent in the map or fold call in the sequential execution, as expected.

5. Related work

Several attempts have been made to provide parallel libraries according to the "minimal disruption" principle. SkeTo [10] implements a data parallel only skeleton library in C/C++. The data parallel skeletons are implemented through library functions that are seamlessly called within user sequential code, much in the style of our `Parmap`. This framework can be hardly compared with `Parmap`, due to the fact the programs are written in imperative style. STAPL [11] is a C++ library providing "parallel containers" with adaptive implementation of standard parallel algorithms that can be used on both multi-cores and distributed architectures. The "minimal disruption" problem is addressed here by providing container classes and algorithms as much as possible close to the one in the standard C++ library.

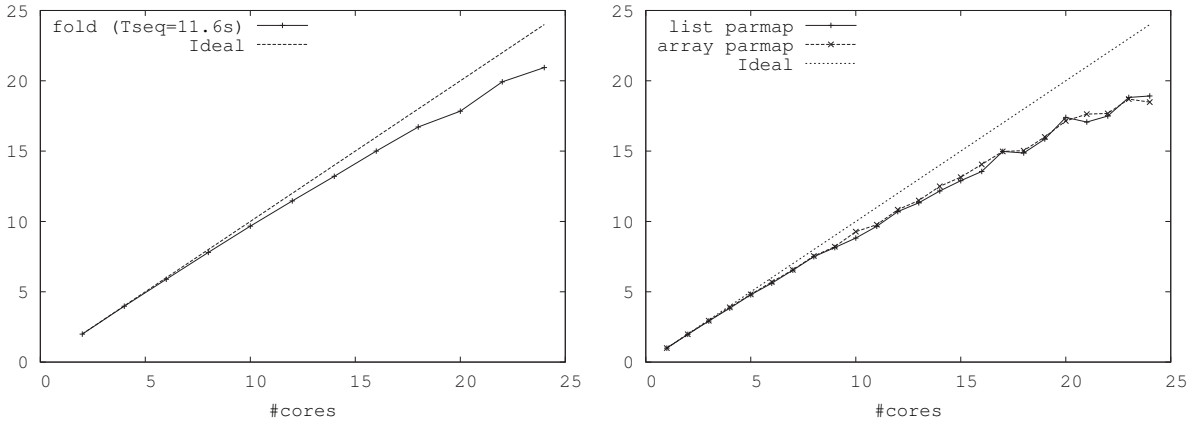

Figure 4: Sample `parfold` speedup (left) and comparison lists vs. arrays (right) (AMD Opteron 6176 x2)

Other functional programming frameworks based on the concept of algorithmic skeletons have been designed in the past. OCamlP3l [12] used a quite different template based implementation and targets TPC/IP POSIX distributed architectures. It requires a substantially different programming style with respect to the one guaranteed by `Parmap`. The same remark apply to the Functory system [13], which is quite similar in spirit to OCamlP3l. Parallel Haskell [14] and Erlang [15] both provide nice parallel programming frameworks targeting multi-cores and distributed architectures. Parallel Haskell, in particular, provides suitable ways to define skeletons by specifying parallel evaluation *strategies* for higher order functions modelling the skeleton functional semantics. Although specifying a parallel evaluation strategy only requires minimal syntactic modifications to the sequential code, it actually requires some extra knowledge from the programmer with respect to the kind of knowledge required to the `Parmap` programmer. In particular, the Haskell programmers are required to figure out which is the better parallel strategy to use among the ones available, whereas the `Parmap` programmer is just required to substitute a function call and all the details related to the parallel implementation are transparently handled by the `Parmap` implementation. While the Haskell approach may support more general parallel patterns, our approach supports a smoother integration within the "normal" (sequential) programming style of programmers.

Map and reduce have been considered as convenient structured parallel programming constructs since a very long time. Backus individuated them as convenient higher order functions in his Turing award lecture [16]. Although not explicitly targeting parallel execution, the algebra of programs defined in Backus's masterpiece have been used by a number of researchers to support development of frameworks providing parallel map and reduce. Bird and Meertens elaborated a complete theory supporting concepts related to parallel execution of map and reduce in the '80 [17]. This theory has been used in SkeTo to apply automatic program transformations improving program performance and, although never mentioned in their papers, is the reason of the success of Google's mapreduce [18].

6. Conclusions

`Parmap` is a minimalistic library allowing to exploit multi-core architecture for OCaml programs. It has been designed with the goal of providing parallel map and reduce to OCaml programmers in a fairly natural way, such that the "minimal disruption" principle stated by Cole in his skeleton manifesto paper is enforced. In fact, in order to use `Parmap`, it is sufficient to substitute the calls to `List` functions with calls to the equivalent `Parmap` functions. The clean and efficient implementation of `Parmap` is such that nearly optimal speedups are achieved on state-of-the-art multi-core architectures when suitable grain computations are parallelized. The full source code of the `Parmap` library is available under the LGPL licence from `http://gitorious.org/parmap`, and is now also incorporated in the GODI installation system for OCaml librairies.

The authors would like to thank Paul Vernaza, François Berenger and Pierre Chambart for stimulating discussions

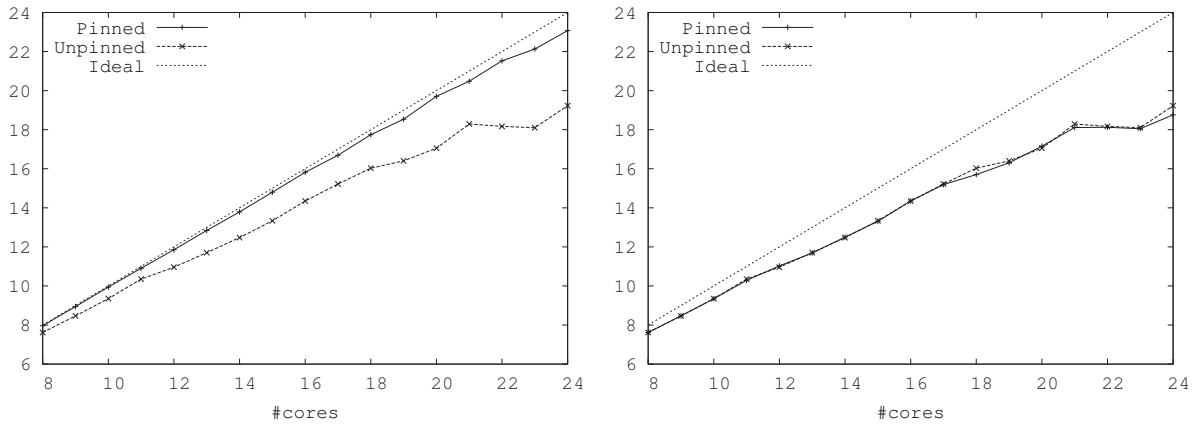

Figure 5: Effect of pinning: speedup in case of coarse grain/scalable (left) vs. finer grain/non scalable (right) applications (AMD Opteron 6176 x2)

about Parmap, Jérôme Vouillon for his contributions to the code that greatly improved its efficiency, Pietro Abate for help with the build system, and Jérôme Maloberti for creating the package for the GODI OCaml distribution system.

References

[1] T. Mattson, B. Sanders, B. Massingill, Patterns for parallel programming, Addison-Wesley Professional, 2004.
[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape, Commun. ACM 52 (10) (2009) 56–67.
[3] D. Goswami, A. Singh, B. R. Preiss, From design patterns to parallel architecture skeletons, Journal of Parallel and Distributed Computing 62 (4) (2002) 669–695.
[4] M. Cole, Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.
[5] M. Aldinucci, M. Danelutto, P. Dazzi, Muskel: an expandable skeleton environment, Scalable Computing: Practice and Experience 8 (4) (2007) 325–341.
[6] P. Ciechanowicz, H. Kuchen, Enhancing Muesli's Data Parallel Skeletons for Multi-Core Computer Architectures, in: Proc. of the 10th Intl. Conference on High Performance Computing and Communications (HPCC), IEEE, Los Alamitos, CA, USA, 2010, pp. 108–113.
[7] R. Di Cosmo, Z. Li, S. Pelagatti, P. Weis, Skeletal parallel programming with ocamlp3l 2.0, Parallel Processing Letters 18 (1) (2008) 149–164.
[8] J. Enmyren, C. W. Kessler, SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems, in: HLPP 10: Proceedings of the fourth international workshop on High-level parallel programming and applications, 2010, pp. 5–14, baltimore, Maryland, USA.
[9] M. Leyton, J. M. Piquer, Skandium: Multi-core programming with algorithmic skeletons, in: Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing, IEEE, Pisa, Italy, 2010, pp. 289–296.
[10] H. Tanno, H. Iwasaki, Parallel skeletons for variable-length lists in sketo skeleton library, in: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 666–677.
[11] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, L. Rauchwerger, A framework for adaptive algorithm selection in stapl, in: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, ACM, New York, NY, USA, 2005, pp. 277–288.
[12] M. Danelutto, R. Di Cosmo, X. Leroy, S. Pelagatti, Parallel functional programming with skeletons: the OCamlP3L experiment, in: Proceedings ACM workshop on ML and its applications, Cornell University, 1998.
[13] J.-C. Filliâtre, K. Kalyanasundaram, Functory: A Distributed Computing Library for Objective Caml, in: Trends in Functional Programming, Madrid, Spain, 2011.
[14] P. W. Trinder, R. F. Loidl, Hans-WolfgangPointon, Parallel and distributed Haskells, Journal of Functional Programming 12 (4&5) (2002) 469–510.
[15] F. Cesarini, S. Thompson, Erlang programming: a concurrent approach to software development, OŔeilly Media, 2009.
[16] J. W. Backus, Can programming be liberated from the von neumann style? a functional style and its algebra of programs, Commun. ACM 21 (8) (1978) 613–641.
[17] D. B. Skillicorn, The Bird-Meertens formalism as a parallel model, in: Software for Parallel Computation, volume 106 of NATO ASI Series F, Springer, 1993, pp. 120–133.
[18] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113.